Implementing a semi-causal domain-specific language for context detection over binary sensors

Nic Volanschi, Bernard Serpette, Charles Consel

Implementing a Semi-causal Domain-Specific Language for Context Detection over Binary Sensors

Nic Volanschi
Inria Bordeaux
Talence, France
eugene.volanschi@inria.fr

Bernard Serpette
Inria Bordeaux
Talence, France
bernard.serpette@inria.fr

Charles Consel
Bordeaux INP & Inria Bordeaux
Talence, France
charles.consel@inria.fr

Abstract

In spite of the fact that many sensors in use today are binary (i.e. produce only values of 0 and 1), and that useful context-aware applications are built exclusively on top of them, there is currently no development approach specifically targeted to binary sensors. Dealing with notions of state and state combinators, central to binary sensors, is tedious and error-prone in current approaches. For instance, developing such applications in a general programming language requires writing code to process events, maintain state and perform state transitions on events, manage timers and/or event histories.

In another paper, we introduced a domain specific language (DSL) called Allen, specifically targeted to binary sensors. Allen natively expresses states and state combinations, and detects contexts on line, on incoming streams of binary events. Expressing state combinations in Allen is natural and intuitive due to a key ingredient: semi-causal operators. That paper focused on the concept of the language and its main operators, but did not address its implementation challenges. Indeed, online evaluation of expressions containing semi-causal operators is difficult, because semi-causal sub-expressions may block waiting for future events, thus generating unknown values, besides 0 and 1. These unknown values may or may not propagate to the containing expressions, depending on the current value of the other arguments.

This paper presents a compiler and runtime for the Allen language, and shows how they implement its state combining operators, based on reducing complex expressions to a core subset of operators, which are implemented natively. We define several assisted living applications both in Allen and in a general scripting language. We show that the former are much more concise in Allen, achieve more effective code reuse, and ease the checking of some domain properties.

GPCE '18, November 5–6, 2018, Boston, MA, USA

ACM ISBN 978-1-4503-6045-6/18/11…$15.00
https://doi.org/10.1145/3278122.3278134

CCS Concepts • **Software and its engineering** → **Domain specific languages**; • **Human-centered computing** → *Ubiquitous computing*; • **Hardware** → *Sensor applications and deployments*; • **Information systems** → Stream management; Data streaming;

Keywords domain-specific languages, binary sensors, context awareness

ACM Reference Format:
Nic Volanschi, Bernard Serpette, and Charles Consel. 2018. Implementing a Semi-causal Domain-Specific Language for Context Detection over Binary Sensors. In *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '18), November 5–6, 2018, Boston, MA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3278122.3278134

1 Introduction

Sensors are the foundation of pervasive computing applications: they are embedded in handheld and wearable devices, in many embedded systems, and they also are the keystones of smart spaces. Some pervasive application domains mostly or exclusively rely on *binary* sensors, that is, sensors producing only values of 0 and 1.

The smart homes domain for example heavily relies on binary sensors, for several reasons. First of all, at the physical level, many commodity sensors used for equipping smart homes are binary by nature: motion sensors signal presence or absence in a given area; contact sensors are attached to home objects such as doors or drawers to signal their open and closed states; pressure mats are placed at strategic locations in smart buildings to detect walking and trigger specific automata, etc. On a more conceptual level, smart homes are mostly interested in detecting environment situations and interactions which are conceptually detected by binary sensors, even if the underlying physical sensors may not be binary. Thus, situations such as an over-heated or under-heated area are typically detected by temperature sensors passing over a threshold values. Interactions of a user with the environment such as preparing a meal or doing cleanup are usually detected by smart switches attached to strategic devices such as cookers or washing machines, also associated with some threshold values, to signal the states of using, or not using, the corresponding devices.

Consequently, several research works studied algorithms based on binary sensors for activity recognition [15, 16, 18] in smart homes, but also in other application domains such as object tracking [6] or smart manufacturing [23].

As a direct consequence of their particular value domain, binary sensors share the notion of *state*, spanning over a *time interval*. Associated to these notions, other derived notions such as the *current state* or *temporal constraints* over states, like a state lasting at least/at most a given time, are of interest in most applications. Furthermore, real applications rely on multiple binary sensors, which brings into play various notions of state combination, such as several states being concomitantly or alternatively true, or *temporal orderings* between their time intervals, like sequencing, overlapping, or containment. This rich set of concepts and their possible combinations strive for being supported by a domain-specific approach. Indeed, in absence of specific support, implementing context detection logic in general programming languages is highly technical, tedious, and error prone. Specifically, when developers have to repeatedly implement these concepts and combinations in different applications, they may introduce subtle differences that may translate to incoherences in behavior and bugs.

The problem is exacerbated when many different variations of (even small) context detection logics have to be developed and deployed. This is usually the case when pervasive applications are intimately intertwined with user lives, which is the case of every smart home application. Indeed, services involving users have to be adapted to user profiles and preferences, leading to many variations of the context detection logic. For instance, an application detecting a daily activity such as meal preparation has to closely match the way each person performs this activity, which may involve using various appliances, according to different temporal patterns, and within customizable time slots. It has been shown indeed that customizing assistive services in a smart home to the users abilities plays a key role towards achieving acceptance of these technologies [7]. This stringent need for customization entails that scalability in terms of supported needs can be achieved only if an important number of context-aware services and variations thereof can be developed easily. In particular, when talking about variations of a given service, it is clear that the ability to reuse logic is also a key requirement.

In a companion paper [22], we introduced a domain-specific language (DSL) for developing context detection logic over binary sensors that is able to fulfil these requirements effectively. Our language, called Allen, natively supports the concepts of state and state combinators, and also the associated concepts of current state, temporal constraints and orderings. Both raw information coming from sensors and more refined *contexts* computed by applications are modeled as boolean functions of time, also called signals. New contexts are thus computed by applying operators to signals. The set of predefined operators can be extended by user-defined operators. Like native operators, user-defined operators apply to a given number of signals, and may also be parameterized by integer constants such as delays or thresholds.

That paper introduced the concepts of the Allen language, its operators and their semantics, presented a prototype interpreter, and compared it to the Complex Event Processing (CEP) language called Esper. In turn, no details were given about how operators were implemented in the interpreter.

This paper briefly introduces the Allen language in order to be self-contained, then focuses on the challenges involved in its implementation. Furthermore, it situates this DSL with respect to the use of general programming languages for programming context-aware services. Our contributions are:

- We motivate and evaluate our language by comparing the implementation of real assistive services in our language and in a general scripting language. Programs in Allen are much shorter, facilitate reuse, and simplify checking some specific behavior properties.
- We show why implementing the Allen operators is not straightforward. Difficulties mainly result from the online evaluation of semi-causal operators and non-strict operators.
- We describe implementation techniques able to solve these difficulties and implement an online evaluator consuming streams of events coming from sensors and producing streams of detected contexts. Our new implementation also contains a compiler that produces Perl code to be executed in conjunction with a runtime.
- We also show some optimization techniques for simplifying the implementation of the language and for decreasing the memory used at runtime.

Before introducing our DSL in section 3, we start by a motivating example that will be used to illustrate its main features and advantages, in the next section.

2 Motivating Example

As a motivating example, let us consider a simple but real smart home assistive application called DoorAlert, which aims at detecting security situations related to the entrance door left open. From the user's point of view, such an application is relevant because the safety needs have been identified as one of the areas that can be addressed in a promising way by technology, for different categories of users, including normally aging seniors, or individuals with special needs such as Alzheimer's disease [3]. From the perspective of our discussion, this application is interesting because it is a typical application relying on several binary sensors and involving different temporal constraints and orderings. Furthermore, it illustrates the needs for customization that are common to assistive applications, because different versions of this application must be developed for different user profiles.

More specifically, DoorAlert detects the situation when the entrance door has been left open *and* unattended for at least some given time T. This may arrive due to an oversight of the home occupant, and is intended for seniors having a moderate decline of short-term memory. When the situation is detected, the application sends a reminder to the user to close the door. However, when the user stands in the open door, for instance having a discussion with somebody outside the home, the application should not signal this situation, because the door is not unattended in this case. The application relies on two physical binary sensors: a contact sensor attached to the entrance door, and a motion sensor covering the area inside the home near the entrance door.

A variation of this application, called DepartureAlarm, is intended for users presenting a risk of anxiety or disorientation which may lead to a night wandering episode, such as individuals with Alzheimer's disease. The DepartureAlarm application detects the situation when the door has been left open for at least some time T during the night, assuming that an appropriate time slot has been defined for the night time. In this version, only one physical sensor is used, the contact sensor on the door, but its information is crossed with a simple "software sensor" detecting night time. When the situation is detected, an alarm is sent to a caregiver, for example taking the form of an SMS.

2.1 Application Development in a GPL

Let us examine how such pervasive applications can be developed using a general programming language (GPL). We choose Perl here, as this language is frequently used in practice for processing textual sensor logs (possibly pipes, in case of online sensor processing) in a variety of formats, either record-oriented (e.g., CSV) or semi-structured (e.g., JSON). This is also why our language is compiled to Perl.

Due to space restrictions, we will detail here only the DoorAlert application. Its hand-coded Perl version implements the automaton in Figure 1. Note that for the base version of DoorAlert, the dashed elements in the figure must be ignored. The automaton contains four states that encode all the possible states of the two sensors, one being open or closed, and the second signalling a presence or an absence near the door. The initial state is usually the one in which the door is closed and the user is not nearby. Transitions are made whenever one of the sensors signals a new value. Some transitions contain, besides the event condition, an action such as initiating a timer (shown as "t=0"), cancelling a timer, or raising an alert. The expiration of a timer can also cause a transition, under the condition "t=T?".

This automaton can be implemented in a straightforward way by the Perl code in Figure 2.[1] The code consists of an event processing loop handling each incoming event, and extracting from it relevant items via regular expression matching, such as its timestamp, the sensor name, and its reported value. The body of the loop must: handle all relevant events; maintain state as necessary; manage timers; and record event histories (in the hash table `lastval`).

In our case, the state is one of the four states in the automaton, which have been encoded using two boolean variables, `opened` and `present`. The timer is implemented by a variable, `unwatchedts`. If the timer is active, this variable holds the timestamp when the timer was triggered, otherwise, it is undefined (i.e., has the predefined `undef` value in Perl). Timeouts are detected at the beginning of the loop body by checking if the period of time since the timer is active up to the current event is at least T. Note that this code excerpt is simplified in that it checks timeout only when sensor events happen in the house. In the complete code, special timeout events are inserted into the stream of sensor events, and therefore, timeouts are detected as soon as they happen, without waiting for a new sensor event. This does not change the logic of the loop body; it only simplifies our presentation of the code. Other simplifications include the handling of options from the command line, such as the value of T. Overall, the shown excerpt has 30 lines, while the complete code has 65 lines, excluding user-defined subroutines such as `period`, which may be implemented in a library module.

From this simple example, one can see that the loop body is a generic event handler, able to recognize all relevant events and handle them by performing the necessary transitions and actions. It is well known that this coding structure does not scale well to more complex cases, involving a greater number of sensors and temporal constraints and orders. In such cases the code becomes tedious to program and error prone, a situation known as the *callback hell* [4]. Maintaining an accurate automaton model along with the code (or deriving the code from an automaton model, if appropriate tool support is available) alleviates the problem to some extent, but on the automaton model as well, they are many subtleties to be mastered. For instance, even in our simple case, one must take care to initiate the timer on all transitions towards the `AwayOpen` node, and only on these, and to reset it on all transitions leaving that node other then the self-loop actioning the alert. Failing to do so may lead to an "escaped timer" which could trigger a false alert when some specific paths are taken. The problem only becomes more complex when several timers are involved in the automaton.

In general, this automaton-based event handling paradigm forces developers to compile high-level temporal constraints and orderings to low-level concepts such as state transitions and timers. This repeated, manual compilation of common domain patterns bears the risk of introducing subtle variations between the various instances. Furthermore, simple properties in the application domain must be repeatedly checked on the manually compiled automata. For example,

[1] Note that in Perl, scalar variable names start with `$`, and initially have value `undef`. Hash table names start with `%`, but extracting a value from a hash table `%x` is written `$x{key}`. Comments start with #. Numbers are compared with == and strings with eq.

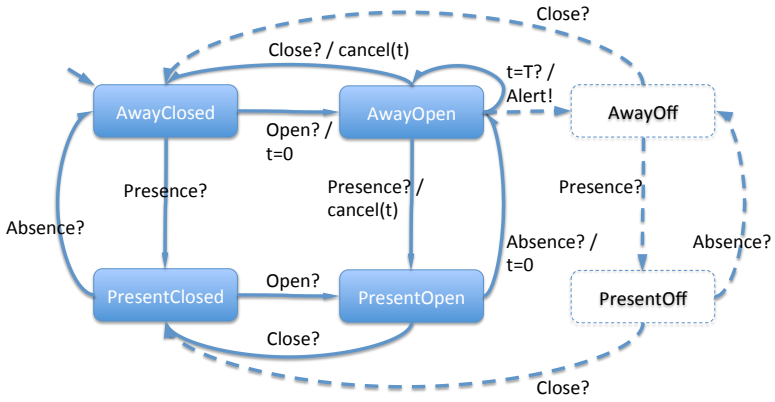

Figure 1. Automaton of the DoorAlert application (solid lines), and DoorAlert1 application (solid+dashed lines).

```
my $T; # timeout for unwatched door (in minutes)
my %lastval = (); # hash table with last value of each sensor
my $opened; # boolean state of the door
my $present; # boolean state of motion detector
my $unwatchedts; # begin timestamp of unwatched state

while (<>) { # for every line read on standard input (= 1 event)
  # pattern match relevant elements from incoming event
  if (my ($date, $val, $sensor, $ts) = /.../) {
      if (defined($unwatchedts) &&
          period($unwatchedts, $ts) >= $T * 60) {
        print "[$date] door unwatched since $unwatchedts\n";
        $unwatchedts = undef;
      }
      if ($sensor eq 'Door' && $val != $lastval{'Door'}) {
        $opened = $val;
        if ($opened == 1 && $present == 0) {
          $unwatchedts = $ts; # set timer on open & absent
        } elsif ($opened == 0 && $present == 0) {
          $unwatchedts = undef; # cancel timer on close & absent
        }
      } elsif ($sensor eq 'Hall' && $val != $lastval{'Hall'}) {
        $present = $val;
        if ($present == 0 && $opened == 1) {
          $unwatchedts = $ts; # set timer on leave & opened
        } elsif ($present == 1 && $opened == 1) {
          $unwatchedts = undef; # cancel timer on come & opened
        }
      }
      $lastval{$sensor} = $val;
  }
}
```

Figure 2. Hand-coded Perl implementation of DoorAlert

the fact that the automaton in Figure 1 raises an alert each time the door is left unattended for time T, and only in this situation, has to be checked by (1) inspecting all the paths in the automaton leading to the alert-triggering transitions to ensure that the timer is appropriately set, and (2) ensuring that the union of the conditions for taking alert transitions exactly covers the given situation.

2.2 Application Evolution in a GPL

It is also instructive to examine what kind of code reuse can be achieved when a new variation of an application implemented in a GPL has to be created. Continuing our example in Figure 1, let us assume that a new version of this application, called DoorAlert1, has to be developed, which signals the same situation of an unattended door, but at most once per door opening. Indeed, the main version of DoorAlert may raise an alert several times for a long door opening, any time when the user disappears from the entrance hall for time T or more. The DoorAlert1 variation is not meant to upgrade DoorAlert, but offer another choice for people having a greater degree of autonomy. For them, the designer assumes that once the user came to the Hall and inspected the situation without closing the door, it is no more necessary to alert again about this situation until the door is closed. As already mentioned, such a close matching between application behavior and user profile is mandatory to avoid user fatigue by over-notification, which could cause the rejection of the assistive technology.

From the technical point of view, this variation can be implemented as shown in the same Figure 1, by adding the dashed states and transitions. As a particular case, the self-loop on state `AwayOpen`, figured by a solid line is *replaced* by the transition towards state `AwayOff`. This has the effect of disabling the alert until the door is closed. When this happens, the automaton transitions to one of the states `AwayClosed` or `PresentClosed`, depending on user presence, thus reenabling the alert. In the Perl code of DoorAlert1 (not shown), this could be implemented by adding a flag `enabled`, initially set to 1, that is reset when the alert is triggered, and set back whenever the door is closed. This flag could then condition both actions of arming and cancelling the timer.

Whichever level we are considering, either the automaton model or the GPL code, it is difficult to reuse the behavior of DoorAlert into DooAlert1, other than copying and pasting

the behavior, and adapting the new copy. This is because the behavior is not reused as a black box, but rather as a *white box*. Thus, in the automaton, the self-looping transition has to be deleted and replaced with another one. In the GPL code, modifications have to be performed at several program points. This copy/paste reuse has well known inconveniencies, such as duplicating maintenance tasks. For instance, if the base version is evolved for any reason, say to take into account a time slot, all the modifications have to be reported on the new copy. Another inconvenient of this white-box reuse is that domain properties that were checked on the initial version DoorAlert must be checked again on DoorAlert1.

The next section introduces our domain-specific approach to building context-aware applications over binary sensors, which eases their development by enabling very concise programs, enables effective behavior reuse, and simplifies the checking of some domain properties.

3 The Allen Language

Our DSL, called Allen, aims at supporting the development of context-aware applications over binary sensors. For doing so, Allen timely detects a set of contexts over a stream of binary sensor events. Note that Allen only addresses the context detection part of context-aware applications. The actions to be performed when a context is detected are out of the scope of the language. Applications may freely use the output of an Allen program to react as appropriate. We first describe the concepts of the language, and then its syntax and semantics.

3.1 Concepts

Allen models sensors as boolean functions over time, also called signals. Time is discrete in Allen, and more precisely the domain of natural numbers $\mathbb{N}$, starting with 0. For instance, Unix timestamps can be used as time values. The value of the signal for a sensor models the *current value* of the sensors, defined as its last reported value. This definition implies that the current value of a sensor does not change until the opposite value is reported. We therefore assume that the reported values are alternatively 0 and 1 (otherwise, repeated values can be filtered out in a lower layer). Furthermore, we assume that the initial value of each sensor at time 0 is known, and that each sensor reports a finite number of values. These assumptions covers all practical cases of real sensor deployments. Therefore, if a sensor $s : \mathbb{N} \to \mathbb{B} = \{0.1\}$ signals values $s(t_i)_{0 \leq i \leq n}$ where $n \geq 0, t_0 = 0$, and $\forall i \in [0, n) \,.\, t_i < t_{i+1} \wedge s(t_i) = \neg s(t_{i+1})$, the signal corresponding to the sensor can be viewed as a sequence of time intervals $[t_i, t_{i+1})$ over which the current value is $s(t_i)$, plus a last time interval $[t_n, \infty)$, where the current value is $s(t_n)$. Note that the intervals are closed on the left, because the value $s(t_i)$, respectively $s(t_n)$, has just been reported, and opened on the right, because this value lasts until, but excluding, t_{i+1}, when the opposite value $s(t_{i+1})$ is reported, respectively because the value ∞ is not in the domain of s. We call a *state* of s any of these time intervals where its current value is 1. A signal can thus be viewed as a sequence of strictly increasing time intervals, representing its states:

$$s = \{[t, t') | t \in \mathbb{N}, t' \in \mathbb{N}^\infty, t < t' \wedge \forall t'' \in [t, t') \,.\, s(t'') = 1\}$$

such that $\forall [t_1, t_1'), [t_2, t_2') \in s \,.\, t_1' < t_2 \vee t_2' < t_1$, where we note $\mathbb{N}^\infty = \mathbb{N} \cup \{\infty\}$. In the following, we will therefore use signals either as functions of time or as sequences of time intervals, as needed.

Contexts are also modeled as signals in Allen. Context signals are computed by applying operators on sensor signals and on other context signals. There is a rich set of operators in Allen, taking and returning signals. They are detailed next.

```
Prog -> Lib Rules
Lib -> Def*
Def -> "def" id ("[" id ("," id)* "]")? ("(" id ("," id)* ")")?
       "=" Context
Rules -> Context ";" Rules | Context
Context -> "let" id "=" Expr Context | Expr
Expr -> Prod "|" Expr | Prod
Prod -> Comp "&" Prod | Comp
Comp -> Expr1 (">=!"|"<="|">="|"<"|">") Int | Expr1
Expr1 -> "~" Expr1 | "(" Expr ")" | str
  | id ("[" Int ("," Int)* "]")? ("(" Expr ("," Expr)* ")")?
Int -> id | int ("hr" | "min" | "sec")?
```

Figure 3. Grammar of the Allen language.

3.2 Syntax

The syntax of the Allen language is given in Figure 3. The central syntactic construct is the non-terminal `Expr`, defining expressions over signals. The simplest forms of expressions are the name of a sensor, such as `"Hall"` (terminal `str`), or a variable representing a signal, such as `p` (terminal `id`). More complex expressions are built by applying operators to simpler expressions.

The most basic operators are the boolean operators and, or, not, which are pointwise applications of the standard boolean operators on signals. For expressing *time constraints*, there are comparison operators (non-terminal `Comp`) selecting states that are longer or shorter than a given delay, such as `p > 1min`. A delay (non-terminal `Int`) is either an integer constant expressing hours, minutes, or seconds, or a variable valued as such a constant. Finally, as states are modeled as time intervals (where the value of a signal is 1), *temporal ordering* betweens states is expressed using a standard set of operators on time intervals, namely the time interval relations from the Allen logic [2]. These are the 13 possible, mutually exclusive, orderings between two time intervals I1 and I2, as depicted in Figure 4. However, as signals are

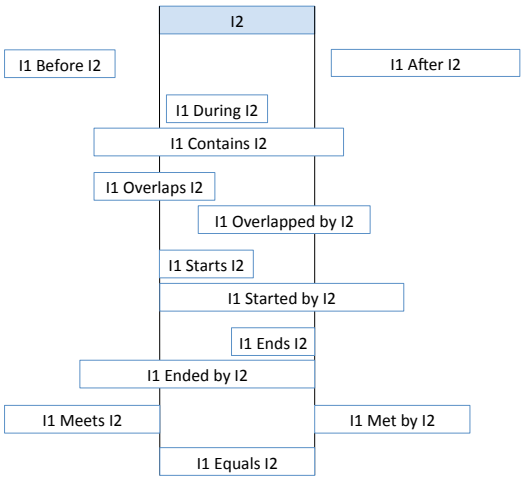

Figure 4. The 13 possible relations between two time intervals.

sequences of time intervals, these relations are generalized to be applicable on signals. The precise semantics of our operators is explained in Section 3.3.

A context in Allen may contain, besides a signal expression, a list of local variable definitions using the `let` construct (non-terminal `Context`). Finally, a program (non-terminal `Prog`) contains a list of contexts separated by semicolons (non-terminal `Rules`), optionally preceded by a library of user-defined operators. A user-defined operator (non-terminal `Def`) associates to a context expression a name, an optional list of parameters, between square brackets, and an optional list of signal operands, between round brackets. Operator parameters must be valued as integer time constants, and signal operands must be valued as signal expressions, as can be seen in the syntax for invoking a user-defined operator (last choice of non-terminal `Expr1`). For instance, the following Allen program defines and invokes an operator which selects the presence states in the shower whose durations are in a given time range:

```
def inrange[min,max](p) = p >= min & p <= max

inrange[5min,1h](Shower)
```

3.3 Semantics

Boolean operators are pointwise applications of the standard boolean operators:

- $(p\&q)(t) = p(t) \wedge q(t)$
- $(p|q)(t) = p(t) \vee q(t)$
- $(\sim p)(t) = \neg p(t)$

Temporal-constraints operators select states of a given signal whose duration fulfil a given constraint with respect to a given delay $T > 0$:

- s < T = $\{[t,t') \in s \mid t' \neq \infty \wedge t' - t < T\}$
- s <= T = $\{[t,t') \in s \mid t' \neq \infty \wedge t' - t \leq T\}$
- s > T = $\{[t,t') \in s \mid t' = \infty \vee t' - t > T\}$
- s >= T = $\{[t,t') \in s \mid t' = \infty \vee t' - t \geq T\}$
- s >=! T = $\{[t,t+T) \mid [t,t') \in s \wedge (t' = \infty \vee t' - t \geq T)\}$

The last operator, >=!, is a stricter version of the operator >= (in the sense that $\forall t, (s >=! T)(t) \rightarrow (s >= T)(t)$), which selects the states of s lasting at least time T, and abbreviates them to a duration of T. This operator is useful to timely signal a state exceeding a given duration, without waiting for the state to be finished. For instance, it is useful to know when a door is left open for 5 minutes, instead of learning that only when the door is closed again, possibly after a long time.

There are other two useful temporal operators that do not constrain the states of a signal:

- $delay[T](s) = \{[t+T,t'+T) \mid [t,t') \in s \wedge t' \neq \infty\} \cup \{[t+T,\infty) \mid [t,\infty) \in s\}$
- $wave[T_1,T_0,T_{end}] = \{[t,t+T_1) \mid t \bmod (T_1+T_0) = 0 \wedge t < T_{end}\}$

Operator delay, as it names suggests, delays a signal for a certain time lapse $T > 0$. Operator wave is a nullary operator, i.e., not taking any signal as an operand; it rather synthesizes a periodic signal itself, which is 1 during time T_1, then is 0 during time T_0, and loops so until time T_{end} is reached. At that time, it just finishes the current cycle and becomes 0 forever. These two operators can be combined to generate a signal which is 1 every day during a given time slot. Such a signal is of great importance for context-aware applications in a smart home, such as daily activity recognizers or comfort automation. For instance, a slot between 8AM and 10AM during 3 days can be defined as follows:

```
def slot[Tstart,T1,T0,Tend] =
  delay[Tstart](wave[T1,T0,Tend])
def breakfast_time = slot[8hr,2hr,22hr,72hr]
```

Time ordering operators generalize the corresponding relations from the Allen logic, in order to apply on multiple time intervals (also called non-convex intervals). Thus, for any operator in Allen logic called R (see Figure 4), operating on two intervals, there is a corresponding operator in our language, called r, operating on two signals p and q. It selects the states in p which are in the relation R with at least one state in q. By making explicits the constraints in the operators R, we obtain:

- $meets(p,q) = \{[t,t') \in p \mid \exists [t',t'') \in q\}$
- $met(p,q) = \{[t,t') \in p \mid \exists [t'',t) \in q\}$
- $eq(p,q) = \{[t,t') \in p \mid \exists [t,t') \in q\}$
- $starts(p,q) = \{[t,t') \in p \mid \exists [t,t'') \in q \,.\, t' < t''\}$
- $started(p,q) = \{[t,t') \in p \mid \exists [t,t'') \in q \,.\, t'' < t'\}$
- $ends(p,q) = \{[t,t') \in p \mid \exists [t'',t') \in q \,.\, t'' < t\}$
- $ended(p,q) = \{[t,t') \in p \mid \exists [t'',t') \in q \,.\, t < t''\}$
- $overlaps(p,q) = \{[t,t') \in p \mid \exists [t'',t''') \in q \,.\, t < t'' < t' < t'''\}$
- $overlapped(p,q) = \{[t,t') \in p \mid \exists [t'',t''') \in q \,.\, t'' < t < t''' < t'\}$

- $during(p, q) = \{[t, t') \in p \mid \exists [t'', t''') \in q \,.\, t'' < t < t' < t'''\}$
- $contains(p, q) = \{[t, t') \in p \mid \exists [t'', t''') \in q \,.\, t < t'' < t''' < t'\}$

Note that the Before and After relations in Allen logic do not have a meaningful counterpart in our language. For example, selecting states in p that are Before a state in q comes to selecting almost all states in p. Thus, we left out these two operators.

There are two additional operators that have been added later on to the Allen logic [12], that we have adopted in our language, too. They select the states of a signal q in which signal p holds all the time, respectively occurs at least once:

- $holds(p, q) = \{[t, t') \in q \mid \forall t'' \in [t, t') \,.\, p(t'')\}$
- $occurs(p, q) = \{[t, t') \in q \mid \exists t'' \in [t, t') \,.\, p(t'')\}$
- $occurred(p, q) = \{[t'', t') \mid [t, t') \in q \wedge \exists t'' \in [t, t') \,.\, p(t'') \wedge \forall t''' \in [t, t'') \,.\, \neg p(t''')\}$

Note that we added a third operator, called occurred, which is a stricter variant of the occurs operator: instead of selecting the whole states of q where p occurs, it selects only the ending sub-intervals of each such state where p has already occurred. This operator is useful to signal the precise moments when p occurs *for the first time* in each state of q.

4 Qualitative Evaluation

Using the above operators, it is easy to write the context detection logic for our running example, as follows:

```
def dooralert[T] = ("Door" & ~"Hall") >=! T
```

The DoorAlert application simply selects the combined states where the entrance door is open and at the same time there is no motion in the Hall, combined states which furthermore exceed a duration of T. All this is simply done using three signal operators: negation, conjunction, and lower bound time constraint. This signal expression is encapsulated in a user-defined operator using the `def` construct, parameterized with the timer delay. No automaton has to be designed by the developer to recognize the corresponding patterns of events sequences, and there is no need to explicitly set timers and handle timeouts. All these low-level details are delegated to the Allen language compiler and runtime. As a result, the Allen program for the application is very concise and self-explaining. This is in sharp contrast with both the automata form in Figure 1 and the GPL form in Figure 2.

Moreover, some domain properties of the program are simple to check, due to the semantics of the involved operators. Thus, it appears clearly that:

- The DoorAlert situation is detected every time the door is open and unattended for at least T. Indeed, the open-unattended situations are all separated by a door closing and/or a presence in the entrance hall. All those lasting at least T are returned by the `>=!` operator.
- Conversely, any state computed by DoorAlert corresponds to a door left open and unattended for at least T. This property is obvious thanks to the staged composition of the involved operators: any state selected by the `>=! T` operator lasts at least T, and any state recognized by `"Door" & ~"Hall"` is an open-unattended door state.
- All the states computed by DoorAlert span exactly time T, start when the door becomes open and unattended, and end time T afterwards. This property is ensured by the semantics of the `>=!` operator.

Finally, the simplicity of code reuse is demonstrated in the DoorAlert1 variant of this application, which reuses the previous user-defined abstraction, simply encapsulating it in a new operator application.

```
def dooralert1[T] = occurred(dooralert[T], "Door")
```

The `occurred` operator ensures, by its semantics, that at most one alert is raised per door opening state. Thus, as opposed to the copy/paste reuse in the Perl code or automata solutions, there is no modification to be done here on the reused code; the new DoorAlert1 abstraction is obtained by composing the reused abstraction, as is, with some new domain logic. The parameterization of the reused abstraction is also propagated to the new one.

Other forms of reuse could be expressed, by making the reusable abstraction more generic. For instance, let us assume that the DoorAlert application must be installed in a home with two entrances, both equipped with a contact sensor and a motion detector. It is easy to pass these sensors as operands of the DoorAlert abstraction, then instantiate the abstraction twice, and define their new composition using an `or` operator:

```
def dooralert[T](door, hall) = (door & ~hall) >=! T
# Instantiate twice in a context:
dooralert[10min]("FrontDoor", "Hall") |
dooralert[5min]("BackDoor", "Cellar")
```

The DepartureAlarm application can similarly be encoded in a concise and self-explaining form, by reusing the user-defined `slot` operator defined in Section 3.3, as follows:

```
def departurealarm[T,Tstart,T1,T0,Tend] =
  during("Door" >=! T, slot[Tstart,T1,T0,Tend])
# Instantiate for T=2hr and night slot from 10PM to 6AM:
departurealarm[2hr,22hr,8hr,16hr,72hr]
```

These examples show that our language brings some advantages in terms of conciseness, reusability, and the possibility to check domain properties. The next section discusses the challenges of implementing our language.

5 Implementation

Implementing the Allen language would be rather straightforward if evaluation of contexts could be done in batch mode. That is, if all sensor events accumulated during their deployment would be stored in a database, all the temporal

ordering operators could be implemented by simple relational queries, directly translating their set-based semantics. However, implementing the Allen operators in an online setting, i.e. handling events continuously as they are produced by the deployed sensors, is far from trivial. We first explain why the online setting is important for the context detection, and the difficulties that arise from that. The rest of this section details some implementation techniques that can be used to solve these difficulties, and furthermore some optimization techniques decreasing the cost of implementing the language, or improving the efficiency of its implementation in terms of memory consumption. At the end of the section, we briefly describe our prototype implementation, which integrates all these techniques.

5.1 Online Context Evaluation

Timely detecting contexts based on the streaming data coming from sensors is a key feature in context-aware applications in various domains. For instance, in context-aware applications in the Internet Of Things (IoT), real-time processing is considered essential due to the volume of data [17]. Also, many real-world applications that focus on addressing the needs of a human require information about the activities being performed by the human in real time [15]. However, the online processing of incoming events constitutes a real challenge for the set of operators described in the previous section. Indeed, when examining the semantics of most operators, one can see that they are *semi-causal*. That is, the current value of the signal computed by some operator at time t sometimes depends on events in the operand signals that happen in the future, at times strictly greater than t. We use this term as opposed to operators whose value at time t only depends on past and present value of its operand signals, which are called *causal* [14]. For instance, the boolean negation operator is causal, because the value of the expression $\sim p$ at time t only depends on the current value of p at time t. On the contrary, the $>=!T$ operator is semi-causal, because the meaning of the expression $(p >=!T)(t)$ sometimes depend on values $p(t')$ with $t' > t$. To see why this is so, recall from Section 3.3 that this operator selects the states of p lasting at least time T, and keeps only their beginning of length T. Figure 5 illustrates the signals computed for the DepartureAlarm application, featuring the operator Door >=! $2hr$. We can see that:

- when the current value of Door is 0, e.g., Door(18) = 0, it is known in real time that: (Door >=! $2hr$)(t) = 0, because there is no state to select at this time;
- when a state of Door starts because the door is open, for instance at times 14 and 23, its duration is not yet known, so it is unknown yet whether this state will be selected, hence the value of (Door >=! T)(t) is unknown.

These two situations are represented differently in the signal of the expression Door >=!$2hr$: as a solid line when the value is known in real time, and as a dashed line when the value is only computed a posteriori. Note that the period of incertitude ends either when the door is closed in less than $T = 2hr$ (e.g. at $t = 15$), or otherwise after time $T = 2hr$, when the door opening is known to last at least T (at $t = 25$). At this precise time ($t = 25$), the signal is both computed a posteriori as 1 over the last period of T, and resets to 0.

Let us now focus on the whole expression of the DepartureAlarm application, which consists in applying the `during` operator to the signal above and to the night slot: *during*(Door >=! $2hr$, Night). The second operand, the night slot, is always known in real time, as it only depends on the current time of the day, which can be considered as a sort of software sensor. As opposed to that, we saw that the first operand, the expression Door >=! $2hr$, becomes unknown during two time intervals, [14, 15) and [23, 25). A naive implementation of the Allen language would blindly propagate these incertitude intervals to the `during` expression. However, recall that the semantics of this operator (see section 3.3) is to select the states of its first operand that are properly included in some state of its second operand. Therefore, the value of `during` is always 0 whenever its second operand is 0, *regardless* of its first argument's value (we may say that the operator is *non-strict* in the first argument). In particular, this is the case during the whole interval [10, 22). Hence, the incertitude on the first operand during [14, 15) does not have to be propagated: the value of `during` is still 0 and known in real time. This explains the solid line on this signal at that period. On the other hand, the incertitude on the first argument during [23, 25) does propagate through the `during` operator, because if and only if its value turns out to be 1 during this interval (which happens, actually), this would constitute a state properly contained in the night slot.

This example exposes the two major difficulties for implementing an online evaluator of the Allen language. Firstly, when evaluating a compound expression, the current values of its sub-expressions may be known at some times to be 0 or 1, but may also be unknown at some other times. Furthermore, when such a value becomes known again, the evaluator may have to compute a posteriori some dependent values in the past that now may become known, according to the new information. Thus, the evaluator should manage discontinuous information about values, track dependencies between values, and update the dependencies according to new information when it arrives, all along the chains of dependencies.

Secondly, a careful implementation of the online evaluator can avoid propagating some unknown values, to compute a complex expression as much as possible in real time. Optimizing these cases is important because the given example is not at all marginal. Indeed, most of the operators in the Allen language are semi-causal, and most of them are also

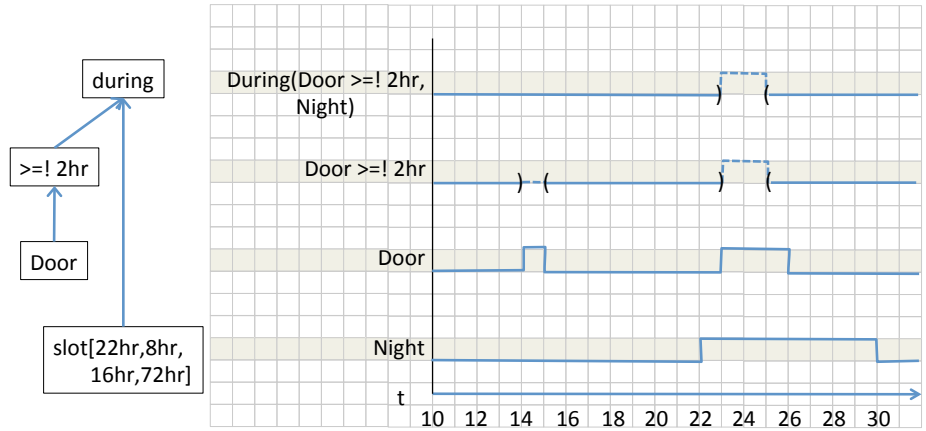

Figure 5. Signals involved in the DepartureAlarm application.

non-strict in at least one argument. Thus, semi-causal operators include all the introduced temporal constraint operators (comparison of a signal to T), and all of the temporal ordering operators inspired from the Allen logic except `met` and `occurred`. We refer the reader to [22] for details, and for a finer characterization of their causality properties. Non-strict operators include `&` (and), `|` (or), `delay`, and all of the temporal ordering operators inspired from the Allen logic. Given this large number of cases, it is crucial to optimize such cases in order to compute composed contexts in real time whenever possible.

5.2 Operator Composition

In order to cope with the first implementation difficulty, that of managing dependencies between possible unknown values, we use an appropriate value domain integrating unknown values, and we introduce adequate data structures for updating dependent values.

Thus, for the current value of any signal, we use an augmented, three-valued domain, $\{0, 1, \perp\}$. A signal has a current value $s(t) = \perp$ whenever its value is unknown with respect to the events happening at times $t' \leq t$. We stress the fact that the expression "the value is unknown" should be understand as "the value could not be computed by the evaluator, using its evaluation strategy", rather than in an absolute logical sense. For instance, the value of the expression $p|{\sim}p$ could in principle be proved to be 1 at any point in time, even when the current value of signal p is unknown, but our evaluator does not attempt at such forms of reasoning.

For managing the dependencies between the values involved in an expression, we build a rooted, acyclic dataflow graph for each context to be evaluated, by parsing the context into an abstract syntax tree, possibly containing shared nodes. Indeed, recall from Section 3.2 that a context is an expression optionally preceded by a list of local variable definitions, using the `let` construct. The different uses of these local variables within the expression lead to shared nodes in the tree. An example of dataflow graph (without shared nodes), corresponding to the DepartureAlarm context, is given in the left part of Figure 5. Each leaf in the graph corresponds either to a sensor such as `Door`, or to a nullary operator such as the `Night` slot. Each internal node in the graph corresponds to an operator applied to one or more signals. Each edge flows from each signal to the operator(s) applied to it, thus creating a data dependency. For instance, two edges flow into the `during` node, corresponding to its two signal operands.

In the simplest case when an operator is causal and strict, its current value is known exactly when the current values of all its operands are known. On the other hand, if the operator is semi-causal, its value may be unknown even when the current values of all its operands are known. Conversely, if the operator is non-strict, its value may be known even when the current values of some of its operands are unknown. Summarizing all the possible cases, at time t, the value of an operator node may be known up to a time $t' \leq t$, and the value of its operands may be known up to times $t'_i \leq t$. For semi-causal operators, t' may be strictly less than all of the t_i; for non-strict operators, t' may be greater than some of the t_i. One might think that in general $t' \leq \max(t_i)$, that is, a signal cannot be computed beyond its most advanced operand. But even this relationship does not hold when considering operators such as `delay[T](p)`, which can tolerate during some time (T in this case) an unknown value of its operand. Thus, there is no general relationship between t' and t'_i. To support such a loose connection between the nodes in the dataflow graph, the evaluator maintains for each node the history of its timestamped values, organized as a list. The

timestamp of each value indicates the moment when the signal switched to this value. Thus, for sensor signals, whenever the sensor produces a new value v at time t, the pair $\langle v, t\rangle$ is appended to the list. For derived signals, which are all three-valued as 0, 1, or $\perp$, some timestamps additionally indicate the moments when the signal became unknown. Thus, when the current value becomes unknown at time t_1, the pair $\langle \perp, t_1\rangle$ is appended to the list. Later on, when the value v becomes known again, the pair $\langle v, t_1\rangle$ is appended to the list. Note that a new (known or unknown) value is appended to the list only if it differs from the last value. The current value of the signal is always the last one, at the end of the list. The list also allows to retrieve older values at any given time t' in the past, by scanning the list backwards until a pair $\langle v, t''\rangle$ is found, with $t'' \leq t'$.

Whenever a set of one or several sensors simultaneously produce values, all the contexts depending on those sensors are updated, if needed. This update is done by a linear pass over its dataflow graph, traversed in topological order to propagate new values from leaves towards the root. The traversal continues only as long as new values are produced by the nodes. Note that a new value may be any form of value change, that is, switching between two known values, switching from a known value to the unknown value, or vice versa.

5.3 Online Operator Evaluation

In order to cope with the second implementation difficulty, that of evaluating online an individual operator applied to some operand signals, we use an incremental algorithm, and we use speculative techniques to integrate in this algorithm optimizations for tolerating unknown values of the operands. We illustrate these technique with the implementation of the `during` operator, which is both semi-causal and non-strict.

As explained above, the value produced by an operator node must be updated any time the value of some of its operand changes, either between two different known values, or between a known and unknown value, in either sense. Moreover, the situations when two operands change simultaneously must be taken explicitly into account, to distinguish between operators requiring simultaneous events, such as `starts`, and operators requiring ordered events, such as `during`. Thus, all these single-event and multiple-events cases have to be carefully studied in order to decide whether the result of the operator has to be updated, either to a known value or to the unknown value.

For the `during(p,q)` operator, only one of these cases concerns a non-causal behavior: the event when, at time t', p changes from 0 to 1 while the current value of q at t' is 1. In this case, signals p and q are examined to see which one is the first to switch back to 0 after t' (recall that operand signals may be known until times $t'_i > t'$). If this information is already available from signals p and q:

- if p becomes 0 before q, the result of during is set to 1;
- if p becomes 0 after or at the same time as q, the result of during is set to 0.

If this information is not yet available, the result of the `during` operator is set to $\perp$. Recall from the previous section that an operator is recomputed whenever new information becomes available on one of its operands. Therefore, the result of the `during` operator will be subsequently recomputed when new events happen on signals p or q. When a value switch will have happened on either p or q, `during` will be recomputed, and will fall in one of the cases above, set to 0 or 1. If set to 1, the subsequent handling of the event when p changes from 1 to 0 will reset the value of `during` to 0. Note, however, that p and q may possibly never change after t'. In this case, the value of `during` will indefinitely stay unknown, which is the right behavior according to the semantics of `during`.

Some other cases concern the non-strict behavior of `during`. All these cases must be found to avoid setting the result of the operator to $\perp$ when unknown operand values can be tolerated. For instance, the case encountered in Figure 5 at time $t = 14$ is when signal p becomes $\perp$ while q is 0. In this case, the value of `during` is set to 0, and the $\perp$ value is not propagated. No less than 10 other cases can be identified for this operator where the $\perp$ value is not propagated.

5.4 Operator Reductions

As can be seen from the example of the `during` operator, implementing an operator is tedious and time consuming. This is no surprise, as this effort factorizes within the implementation of the Allen language efforts that are otherwise repeatedly performed by developers, consisting in encoding domain abstractions in low-level form, paying attention to both correctness issues and optimizations. Nevertheless, we could optimize this implementation effort by observing that some operators can be expressed in terms of others. Thanks to the user-defined operators, we could thus avoid implementing directly the following operators:

```
def occurs(p,q) = q & ~holds(~p,q)
def started(p,q) = occurs(starts(q,p),p)
def ended(p,q) = occurs(ends(q,p),p)
def contains(p,q) = occurs(during(q,p),p)
def overlaps(p,q) = occurs(over(p,q),p)
def overlapped(p,q) = occurs(over(q,p),p)
```

As can be seen, operators `overlaps` and `overlapped` have been reduced to a common native operator called `over`, which is a stricter version of `overlaps`, selecting not the complete states of p that overlap a state of q, but only their ending segments that are superposed with the beginning of a state in q:

- $over(p, q) = \{[t'', t') \mid \exists [t, t') \in p, [t'', t''') \in q \,.\, t < t'' < t' < t'''\}$

We proved these equivalences between operators, based on the semantics of the different operators involved.[2]

5.5 Managing Timers

For implementing temporal operators such as temporal constraints operators (> T and the like), the `delay` operator, or signal synthesisers such as `wave`, the dataflow graph of the contexts containing such operators must be evaluated not only upon events coming from sensors, but also at preset delays, either absolute or relative to other events. For this purpose, the Allen runtime automatically manages timers. When a timer is set to trigger a timeout at time t, a timeout event, timestamped with t, is merged into the streams of events coming from sensors. The online evaluator uniformly handles events in increasing timestamp order, so the timeout event is handled after sensor events which happen before t, before sensor events which happen after t, and simultaneously with sensor events timestamped at t, if any.

For instance, the implementation of the `delay[T](p)` operator sets a timer of T any time the signal p switches its value, and handles each timeout by switching the value of the result. The initial value of the operator is 0. If $p(0) = 1$, an additional timer of T is set at $t = 0$. It is easy to see that this faithfully implements the semantics of the `delay` operator.

Semi-causal and non-strict behaviors must be implemented as for any other operator, by studying all possible impacting events.

5.6 Memory Space Optimization

By default, the complete signal of each node in the dataflow graph is kept. This means that the memory space used by the Allen runtime linearly increases with time: $O(n \times t)$ for an expression with n nodes. If the intermediate signals are not needed, memory space can be saved by dropping elements at the beginning of their lists. Indeed, none of the Allen operators explicitly use events in the past for computing the current value: the current value only depends on present events, and sometimes on future events. For example, the `delay[T]` operator does not need to recall events in the past to reproduce them after time T. Rather, past events have been already handled by creating timers that will trigger after time T, as explained in Section 5.5.

Therefore, the only reason for keeping past events is because some nodes in the dataflow graph may be blocked at some times $t'_i < t$ in the past, where t is the current time, waiting for some future events. Therefore, recomputing their values at time t'_i when those events become available may need the value of their argument signals at time t'_i. It follows that every event older than $min(t'_i)$ may be forgot without disturbing the computation of any node value. We implemented a memory space optimization that periodically erases all obsolete events, defined this way. The period for triggering the forgetting policy is a parameter, which can be varied in order to better amortize its cost.

5.7 Prototype

By using the implementation techniques described above, we implemented a compiler and a runtime for the Allen language. The implementation is written in Perl and amounts to a total of 4200 lines of code.[3] The compiler constitutes about 10% of the code. It parses an Allen program and generates a Perl module containing one Perl subroutine definition for each user-defined operator, and a list of dataflow graphs, one for each context to be evaluated. This Perl module is linked with the runtime, and executed over a stream of incoming events, expressed in a simple textual format based on JSON.

6 Related Work

There are various approaches aiming at simplifying the work of building context-aware systems. A recent survey [1] classifies such works according to the particular development phase that is addressed, such as design, implementation, or testing. As far as programming is concerned, viewed as a part of the implementation phase, most of the existing approaches use general programming languages, and focus on specific mechanisms or extensions for building context-aware systems, such as objects, aspects, features, or agents; a specific line of work concerns context-oriented programming. These approaches focus on how to structure applications for easily incorporating context-dependent features, and how to enable and disable them in specific contexts. The detection logic of the contexts is not specifically supported by a DSL, nor is it specifically focused on streaming sensor events. The few reviewed approaches based on DSLs are aimed at modeling hierarchies of contexts, rather than programming them. A notable exception is the IFTTT DSL, discussed below.

Trigger-action programming (TAP) The IFTTT (If This Then That) language [21] allows end users to express context-aware services as trigger-action rules, where triggers only refer to one event. In particular, triggers mentioning a state such as 'the light is on' refer in fact to the events of state changes, e.g., 'the light gets turned on'. A user study [13] evidenced the fact that specifying services in IFTTT is difficult because the notions of event and state are frequently confused by end users. This study recommends some guidelines for future TAP languages. Taking that work into account, an extension of IFTTT called AppsGate has been recently proposed [8], in which distinct conditional constructs refer to instantaneous events (e.g., as-soon-as, each-time) and to states lasting over a time interval (e.g., if, while). However, conditions must refer to a single event and a single state respectively, which prevents combining events or states between

[2] The proof using the Coq proof assistant is provided with our prototype implementation.

[3] Our open-source prototype is available at https://github.com/NicVolanschi/Allen.

them and with each other. Very recently, the CCBL visual language has been proposed [19] which allows to express triggers combining several states. Combinations are done by graphically nesting state conditions, which is roughly equivalent to using our boolean operators And, Or, and Not between states. There are no other operators combining states. In turn, the language includes actions. An extension of CCBL [20] adds a few operators named after some Allen relations: During, Starts, Ends, and After. However, their semantics is always causal, so it does not exactly correspond to those Allen relations. Note that other Allen relations such that 'Overlaps/Overlapped by' are not covered by CCBL.

Automata Automata are also frequently used for describing context-aware systems reacting to events. We already discussed in the introduction some shortcoming of automata, including the low level of details for managing state relationships and timers, the difficulty of code reuse, and of checking domain properties.

Synchronous languages Synchronous languages such as Esterel [11] propose a textual notation for a higher-level encoding of automata, to describe applications reacting to events, modeled as signals over time. Esterel constitutes a domain-specific, imperative programming language for computing over signals, including an abstraction mechanism called modules, similar to user-defined procedures in general programming languages. Beyond such powerful abstractions, the strength of synchronous languages is that their domain-specific nature enables ensuring strong guarantees about the correctness and real-time behaviour of the programs, including a clean and predictable handling of simultaneity between events. Our Allen-based operators also ensure predictable handling of simultaneous events. There are however a number of issues for handling delays in Esterel [5], which are crucial for implementing our temporal operators. More importantly, synchronous languages allow to encode in a reusable way the causal subset of our operators, but it is an interesting question whether and how our semi-causal operators could be encoded in a synchronous language.

FRP The central concepts of Functional reactive programming (FRP) languages [4, 10] are behaviors (later called signals), which are time-dependent values, and streams of events. The two concepts are inter-convertible using the *hold* and *changes* functions, This paradigm allows to define complex applications by composing event streams and behaviors using an extensible set of operators. Thus, event streams can be filtered, transformed, or merged, and two behaviors can be multiplexed using a third behavior as a continuous condition. The set of operators in the Allen DSL could be implemented as a library of FRP operators. An interesting open question is to what extent the implementation of non-causal operators, involving unknown values, could be simplified using lazy evaluation and the FRP framework. However, a first issue is that lazy streams naturally model future events, that is unknown values becoming known at some point, but our evaluation also deals with the reverse case, when a known value becomes unknown at some point. Also, in most FRP languages, simultaneous events are translated to single events, either by preferring one of the events or by serializing them, in an order specified by the program. In turn, the implementation of most Allen operators critically relies on an explicit handling of simultaneous events.

CEP Complex Event Processing [9] is a paradigm for expressing complex events as patterns of atomic or other complex events. In most CEP languages atomic events are instantaneous, that is, they do not have a duration. Therefore, states must be explicitly coded as pairs of start/end events, leading to complex formulas when combining several states. A detailed comparison of our approach to CEP is in [22].

7 Conclusion

Implementing context-aware applications over binary sensors, using general programming languages, can be tedious and error prone, especially when states from multiple sensors are combined into complex conditions. This hampers the scalability of service development, which is crucial for instance when developing a wide variety of assistive applications for to the needs of different persons and home configurations. We briefly presented the Allen language, which: allows to define context detection components over binary sensors in a very concise way; enables effective code reuse; and simplifies the checking of some domain properties. In turn, implementing the Allen language poses some challenges, mainly due to the presence of semi-causal and non-strict state combining operators. We presented implementation techniques that are able to overcome these difficulties, and further optimization techniques for decreasing implementation cost and making the language runtime more efficient.

In future work, we plan to simplify the implementation even more, by expressing a greater number of operators in terms of other ones, therefore reducing the core subset to a minimal number of native operators. We also intend to explore new compiler optimization techniques for accelerating the execution of such derived operators, to make them execute at comparable speed with the native operators. This is not currently the case, because derived operators are currently translated by replacing their node in the dataflow graph with a subgraph, at each use. Another promising line of work would be to integrate certain common uses of non-binary sensors into the language. For instance, it would be straightforward to incorporate predefined predicates over non binary sensors such as $Temperature > 20°C$, where Temperature is a number-valued sensor, whose conversion into boolean values could be done in an underlying layer, generated by the compiler and deployed with the runtime.

References

[1] Unai Alegre, Juan Carlos Augusto, and Tony Clark. 2016. Engineering context-aware systems and applications: A survey. *Journal of Systems and Software* 117 (2016), 55 – 83. https://doi.org/10.1016/j.jss.2016.02.010

[2] James F. Allen. 1983. Maintaining Knowledge About Temporal Intervals. *Commun. ACM* 26, 11 (Nov. 1983), 832–843. https://doi.org/10.1145/182.358434

[3] Ronald M. Baecker, Karyn Moffatt, and Michael Massimi. 2012. Technologies for Aging Gracefully. *interactions* 19, 3 (May 2012), 32–36. https://doi.org/10.1145/2168931.2168940

[4] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. 2013. A Survey on Reactive Programming. *ACM Comput. Surv.* 45, 4, Article 52 (Aug. 2013), 34 pages. https://doi.org/10.1145/2501654.2501666

[5] T. Bourke and A. Sowmya. 2010. Delays in Esterel. In *SYNCHRON 2009 (Dagstuhl Seminar Proceedings)*, Albert Benveniste, Stephen A. Edwards, Edward Lee, Klaus Schneider, and Reinhard von Hanxleden (Eds.). Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, Dagstuhl, Germany. http://drops.dagstuhl.de/opus/volltexte/2010/2434

[6] Yann Busnel, Leonardo Querzoni, Roberto Baldoni, Marin Bertier, and Anne-Marie Kermarrec. 2011. Analysis of Deterministic Tracking of Multiple Objects using a Binary Sensor Network. *ACM Transactions on Sensor Networks* 8 (2011). https://hal.inria.fr/inria-00590873

[7] Charles Consel, Lucile Dupuy, and Hélène Sauzéon. 2017. HomeAssist: An Assisted Living Platform for Aging in Place Based on an Interdisciplinary Approach. In *Proceedings of the 8th International Conference on Applied Human Factors and Ergonomics (AHFE 2017)*. Springer.

[8] Joëlle Coutaz and James L Crowley. 2016. A First-Person Experience with End-User Development for Smart Homes. *IEEE Pervasive Computing* 15 (May 2016), 26 – 39. https://doi.org/10.1109/MPRV.2016.24

[9] Gianpaolo Cugola and Alessandro Margara. 2012. Processing Flows of Information: From Data Stream to Complex Event Processing. *ACM Comput. Surv.* 44, 3, Article 15 (June 2012), 62 pages. https://doi.org/10.1145/2187671.2187677

[10] Conal Elliott. 2000. Declarative Event-oriented Programming. In *Proceedings of the 2Nd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP '00)*. ACM, New York, NY, USA, 56–67. https://doi.org/10.1145/351268.351276

[11] Abdoulaye Gamatié. 2010. *Designing Embedded Systems with the SIGNAL Programming Language: Synchronous, Reactive Specification.* Springer New York, New York, NY, Chapter Synchronous Programming: Overview, 21–39. https://doi.org/10.1007/978-1-4419-0941-1_2

[12] Malik Ghallab and Amine Mounir Alaoui. 1989. Managing Efficiently Temporal Relations Through Indexed Spanning Trees. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence - Volume 2 (IJCAI'89)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1297–1303. http://dl.acm.org/citation.cfm?id=1623891.1623963

[13] Justin Huang and Maya Cakmak. 2015. Supporting Mental Model Accuracy in Trigger-action Programming. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp '15)*. ACM, New York, NY, USA, 215–225. https://doi.org/10.1145/2750858.2805830

[14] David Janin and Bernard Paul Serpette. 2016. *Timed Denotational Semantics for Causal Functions over Timed Streams.* Research Report. LaBRI - Laboratoire Bordelais de Recherche en Informatique. https://hal.archives-ouvertes.fr/hal-01402209

[15] Narayanan C. Krishnan and Diane J. Cook. 2014. Activity recognition on streaming sensor data. *Pervasive and Mobile Computing* 10 (2014), 138 – 154. https://doi.org/10.1016/j.pmcj.2012.07.003

[16] Fco. Javier Ordóñez, Paula de Toledo, and Araceli Sanchis. 2013. Activity Recognition Using Hybrid Generative/Discriminative Models on Home Environments Using Binary Sensors. *Sensors* 13, 5 (2013), 5460–5477. https://doi.org/10.3390/s130505460

[17] C. Perera, A. Zaslavsky, P. Christen, and D. Georgakopoulos. 2014. Context Aware Computing for The Internet of Things: A Survey. *IEEE Communications Surveys Tutorials* 16, 1 (First 2014), 414–454. https://doi.org/10.1109/SURV.2013.042313.00197

[18] J. Saives, C. Pianon, and G. Faraut. 2015. Activity Discovery and Detection of Behavioral Deviations of an Inhabitant From Binary Sensors. *IEEE Transactions on Automation Science and Engineering* 12, 4 (Oct 2015), 1211–1224. https://doi.org/10.1109/TASE.2015.2471842

[19] Lénaïc Terrier, Alexandre Demeure, and Sybille Caffiau. 2017. CCBL: A Language for Better Supporting Context Centered Programming in the Smart Home. In *The 9th ACM SIGCHI Symposium on Engineering Interactive Computing Systems (PACM on Human-Computer Interaction)*, Vol. 1. Lisbonne, Portugal. https://hal.archives-ouvertes.fr/hal-01534805

[20] Lenaïc Terrier, Alexandre Demeure, and Sybille Caffiau. 2017. CCBL: A new language for End User Development in the Smart Homes. In *Proceedings of IS-EUD 2017*. 82–87. https://pure.tue.nl/ws/files/69763287/IS_EUD2017_extended_abstracts.pdf#page=83

[21] Blase Ur, Elyse McManus, Melwyn Pak Yong Ho, and Michael L. Littman. 2014. Practical Trigger-action Programming in the Smart Home. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '14)*. ACM, New York, NY, USA, 803–812. https://doi.org/10.1145/2556288.2557420

[22] Nic Volanschi, Bernard Serpette, Adrien Carteron, and Charles Consel. 2018. *A Language for Online State Processing of Binary Sensors, Applied to Ambient Assisted Living.* Technical Report submitted for publication. Inria Bordeaux. Available upon request.

[23] Le Yi Wang, Ji-Feng Zhang, and G. G. Yin. 2003. System identification using binary sensors. *IEEE Trans. Automat. Control* 48, 11 (Nov 2003), 1892–1907. https://doi.org/10.1109/TAC.2003.819073